Programming Language Theory

Pseudo-Denotational Semantics for IMP
Looking back, looking forward

- Done: IMP abstract syntax, operational semantics, proofs
- Today: (Pseudo-)Denotational Semantics
- Next: Equivalence
A different approach

Operational semantics defines an interpreter, from abstract syntax to abstract syntax.

- Abstract syntax describes both program and machine
- Metalanguage is inference rules (slides) or ML (imp-ssoper.sml and imp-bsoper.sml)

Denotational semantics defines a compiler (translator), from abstract syntax to a different language with known semantics.

Target language is mathematics, but we’ll make it ML for now.

Metalanguage is mathematics or ML (we’ll show both).
The basic idea

A heap is a math/ML function from strings to integers: \( \text{string} \rightarrow \text{int} \)

An expression denotes (think “can be compiled to”) a math/ML function from heaps to integers.

\[
den(e) : (\text{string} \rightarrow \text{int}) \rightarrow \text{int}
\]

A statement denotes (think “can be compiled to”) a math/ML function from heaps to heaps.

\[
den(s) : (\text{string} \rightarrow \text{int}) \rightarrow (\text{string} \rightarrow \text{int})
\]

Now just define \( den \) in our metalanguage (math or ML), inductively over the source language abstract syntax.
Expressions

\[ \text{den}(e) : (\text{string} \rightarrow \text{int}) \rightarrow \text{int} \]

\[
\begin{align*}
\text{den}(c) & = \text{fn } h \Rightarrow c \\
\text{den}(x) & = \text{fn } h \Rightarrow h \ x \\
\text{den}(e_1 + e_2) & = \text{fn } h \Rightarrow (\text{den}(e_1) \ h) + (\text{den}(e_2) \ h) \\
\text{den}(e_1 \times e_2) & = \text{fn } h \Rightarrow (\text{den}(e_1) \ h) \times (\text{den}(e_2) \ h)
\end{align*}
\]

In plus (and times) case, two “ambiguities”:

▸ “+” from meta language or target language?
▸ \textit{when} do we denote \( e_1 \) and \( e_2 \)?
  ▶ Not a focus of the metalanguage. At “compile time”.
Switching metalanguage

With SML as our metalanguage, ambiguities go away.

But it’s harder to distinguish mentally between “target” and “meta”.

If denote in function body, then source is “around at run time”.

- After translation, should be able to “remove” the definition of the abstract syntax.
- SML doesn’t have such a feature, but the point is we no longer need the abstract syntax.

See imp-denote.sml.
fun denoteExp (e : exp) : (string -> IntInf.int) -> IntInf.int =
case e of
    Num i => (fn h => i)
| Var v => (fn h => h v)
| Plus (e1, e2) => let val de1 = denoteExp e1
    val de2 = denoteExp e2
    in fn h => (de1 h) + (de2 h)
end

(* WRONG: abstract syntax remains at "run time"
   | Plus (e1, e2) => (fn h => ((denoteExp e1) h) + ((denoteExp e2) h)) *)

| Times (e1, e2) => let val de1 = denoteExp e1
    val de2 = denoteExp e2
    in fn h => (de1 h) * (de2 h)
end

(* WRONG: abstract syntax remains at "run time"
   | Times (e1, e2) => (fn h => ((denoteExp e1) h) * ((denoteExp e2) h)) *)
Statements, w/o while

\[\text{den}(s) : (\text{string} \rightarrow \text{int}) \rightarrow (\text{string} \rightarrow \text{int})\]

\[\text{den}(x := e) = \begin{align*}
& \text{fn } h \Rightarrow \text{let val } c = \text{den}(e) h \\
& \quad \text{in fn } v \Rightarrow \text{if } x = v \text{ then } c \text{ else } h v \text{ end}
\end{align*}\]

\[\text{den}(\text{skip}) = \text{fn } h \Rightarrow h\]

\[\text{den}(s_1 ; s_2) = \text{fn } h \Rightarrow \text{den}(s_2) (\text{den}(s_1) h)\]

\[\text{den}(\text{if } e s_1 s_2) = \begin{align*}
& \text{fn } h \Rightarrow \text{if } \text{den}(e) h > 0 \text{ then } \text{den}(s_1) h \text{ else } \text{den}(s_2) h
\end{align*}\]

Same ambiguities; same answers.

Why is the \text{val} \ c = \text{den}(e) h \text{ important in } \text{den}(x := e)\text{.}

See imp-denote.sml
The function denoting a `while` statement is inherently recursive!

Good thing our target language has recursive functions!
fun denoteStmt s =
  case s of
    Skip => (fn h => h)
  | Assign (v, e) => let val de = denoteExp e
    in fn h => let val c = de h
      in fn x => (if x = v then c else h x)
    end
  end
  | Seq (s1, s2) => let val d1 = denoteStmt s1
    val d2 = denoteStmt s2
    in fn h => d2 (d1 h)
    end
  | If (e, s1, s2) => let val de = denoteExp e
    val ds1 = denoteStmt s1
    val ds2 = denoteStmt s2
    in fn h => if (de h) > 0 then ds1 h else ds2 h
    end
  | While (e, s) => let val de = denoteExp e
    val ds = denoteStmt s
    fun loop h = if (de h) > 0 then loop (ds h) else h
    in fn h => loop h
    end
  (* WRONG: diverges at "compile time"
    | While (e, s) => denoteStmt (If (e, Seq(s, While (e, s)), Skip))
  *)
fun denote_prog s = 
  let val d = denote_stmt s 
  in fn () => (d (fn _ => 0)) "ans" 
  end 

Compile-time: val d = denote_prog (parse file) 

Run-time: print (Int.toString (d ()) ^ "\n") 

In between: 
- We have an SML program using only 
  functions, variables, ifs, constants, +, *, >, etc. 
- Doesn’t use Num, Var, Plus, Times, Assign, Skip, Seq, If, While
while (WRONG!)

\[
\text{den}(s) : (\text{string} \rightarrow \text{int}) \rightarrow (\text{string} \rightarrow \text{int})
\]

\[
\text{den}(\text{while } e \ s) = \text{den}(\text{if } e (s ; \text{while } e \ s) \text{ skip})
\]
**while (WRONG!)**

\[
\text{den}(s) : (\text{string} \rightarrow \text{int}) \rightarrow (\text{string} \rightarrow \text{int})
\]

\[
\text{den}(\text{while } e \ s) = \text{den}(\text{if } e (s ; \text{while } e \ s) \text{ skip})
\]

Diverges at compile-time!
The real story

For “real” denotational semantics, target language is mathematics. 
▶ (And we write $[s]$ instead of $\text{den}(s)$)

Example: $[x := e][H] = [H][x \mapsto [e][H]]$

There are two major problems, both due to while:
1. Math functions do not diverge, so no function denotes while 1 skip.
2. The denotation of loops cannot be circular.
The elevator version (ignorable for 740)

For (1), we “lift” the semantic domains to include a special ⊥.

\[ \text{den}(s) : (\text{string} \rightarrow \text{int}) \rightarrow ((\text{string} \rightarrow \text{int}) \cup \bot) \]

- Have to change meaning of \( \text{den}(s_2) \circ \text{den}(s_1) \) appropriately.

For (2), we use \textbf{while} \( e \ s \) to define a (meta)function \( f \) that given a lifted heap-transformer \( X \) produces a lifted heap-transformer \( X' \):

- If \( \text{den}(e)(\text{den}(H)) > 0 \), then \( \text{den}(s) \circ X \).
- Else \( \text{den}(H) \).

Now let \( \text{den}(\text{while} \ e \ s) \) be the least fixed-point of \( f \).

- An hour of math to prove the least fixed-point exists.
- Another hour to prove it’s the limit of starting with \( \bot \) and applying \( f \) over and over (i.e., any number of loop iterations).

Keywords: monotonic functions, complete partial orders, Knaster-Tarski theorem.
Where we are

- Have seen operational semantics and (a taste of) denotational semantics
- Connection to interpreters and compilers
- Useful for rigorous definitions and proving theorems
- Next: Equivalence of semantics
  - Crucial for compiler writers
  - Crucial for code maintainers
- Then: Leave IMP behind and consider functions.